平成２２年７月２０日（火）　担当：秋山　泰
７月２０日　修正

# プログラミング第一　（E）

## 第１２回

- メモリの動的割り当て
  - malloc( ), calloc( ), realloc( )
  - free ( ) ,　メモリリーク
- データ構造の動的割当て
  - 要素１つごとの動的割当て
  - 大きな単位でまとめた動的割当て
- 【補足】main( )の引数： argc, argv

# C言語における変数の種類

## １．自動変数 (auto)

宣言の場所：　関数内。
アクセス範囲：　宣言された関数内からのみアクセス可能。
変数の寿命：　関数が呼ばれるとき作成され、関数終了時に開放。

## ２．静的変数 (static)

宣言の場所：　関数内。"static"という予約語を付加して宣言する。
例） static int data[3] ;
アクセス範囲：　宣言された関数内からのみアクセス可能。
変数の寿命：　関数が終了しても値は残り、次回呼び出し時に有効。

## ３．大域変数 (global)

宣言の場所：　main( ) 関数の外側。特に予約語を付加しない。
アクセス範囲：　プログラム内の全ての関数からアクセス可能。
変数の寿命：　プログラムの終了まで常に有効。
備考：　大域変数を濫用すると副作用の多いプログラムになる。
関数には引数で情報を渡す。結果は戻り値や、引数の
ポインタ渡しで受け取るのが好ましい。
しかし大域変数が「可読性を高める」ことも（前回参照）

# メモリの動的割り当て

配列宣言などでは、コンパイル時点でデータの大きさがわかっている
必要がある。実行時にコンパイル時の領域だけでは不足することがある。

一方、コンパイル時に可能な限り大きな領域を取ろうとすると、
メモリの不足する環境では実行ができないプログラムになったり、
無用に他のプロセスからメモリを奪って悪影響を与える可能性がある。

そこで、読み込むべきデータの大きさ、または計算結果のデータの
大きさを見て、柔軟にメモリ上に領域を確保できないか？

**メモリの動的割り当て**
(dynamic memory allocation)

利点：　作成したプログラムが要求するメモリ量は、解くべき問題に
適切な量となる。

欠点：　プログラムが複雑になり、バグが起きやすい。
メモリリークが起きやすい。
小刻みにメモリの動的割り当てをすると動作が遅くなる。

# malloc ( )関数　　メモリの動的割当て

void *　malloc（size_t　size )

引数で指定するバイト数(size)の領域を、OSに要求して確保する。
取られた領域の先頭番地を指す void* 型のポインタを戻り値として返す。

void* 型ポインタとは、何型を指すのかが規定できない、番地情報のみのポインタであるので（実際にはメモリ番地だけが判れば、この瞬間には用を足す）、何型を指すかきちんと宣言した通常のポインタに代入する。

size_t とは、非負の整数を表す型である。正負を表せる　int 型に比べて２倍までの数値を表せる。（ほぼint型のようなものだと理解して下さい）

例）

```
#define  SIZE   1000000
ptr1 =  malloc (  sizeof(int)  *  SIZE  )
```
int型整数の SIZE個分の領域確保

```
ptr2 =  malloc (  sizeof(node) * SIZE )
```
node型構造体のSIZE個分の領域確保

# calloc ( )関数　　メモリの動的割当て

void*  calloc(size_t  n,  size_t  size);

指定個数(n)だけ、指定バイト数(size)の領域を、OSに要求して確保する。
malloc ()とは異なり、確保された領域は、全て０にクリアされる。
取られた領域の先頭番地を指す void* 型のポインタを戻り値として返す。

calloc( ) は malloc ( )を呼び出している。
malloc( )は原始的であり、全体のバイト数だけを指定するが、
データの型を意識して、何個の何バイト変数だと指定するcalloc( ) を
利用する習慣を付けた方が、やや好ましいのではないか。
領域がゼロクリアされる点も有利であるので、以下ではcalloc( ) を推奨。

```c
#include <stdio.h>
#include <stdlib.h>
void* allocate_memory(int num, int size);

int main(void){
    double temp, *ptr1, *ptr2;

    printf("data1="); scanf("%lf", &temp);
    ptr1 = allocate_memory(1, sizeof(double));
    *ptr1 = temp;
    printf("data2="); scanf("%lf", &temp);
    ptr2 = allocate_memory(1, sizeof(double));
    *ptr2 = temp;

    printf("data1 (at %ld) = %lf¥n", ptr1, *ptr1);
    printf("data2 (at %ld) = %lf¥n", ptr2, *ptr2);
}

void* allocate_memory(int num, int size){
    void *ptr;

    ptr = calloc(num, size);
    if(ptr == NULL) {
        fprintf(stderr, "memory allocation error.¥n");
        exit(1);
    }
    return(ptr);
}
```

malloc( ), calloc( ) 等を利用するときは stdlib.h を。

allocate_memory( ) 関数で double型１つ分の領域確保。ptr1 ポインタに先頭領域のアドレスを代入する。

```
data1=3.14
data2=6.28
data1 (at 5839816) = 3.140000
data2 (at 5839872) = 6.280000
```

sizeof (double) は８バイトだが、管理部分があり、多めに消費する

calloc( )等は割当てに失敗することがあるので、必ず、NULLかどうかを検査すること。（このように自分で関数化するのも好ましい）

# realloc( ) 関数　　メモリの動的再割当て

void*  realloc(void *ptr,  size_t size);

一度確保した領域の容量を変更（拡大・縮小）するための関数。
第一引数は、以前にmalloc( ) または realloc( )関数自体で確保した領域の先頭のポインタを与える。第二引数で指定したバイト数(size)の領域を確保し、その先頭番地を指す void* 型のポインタを戻り値として返す。

注意点１：
　領域の確保に失敗した場合（主に拡大の場合）は、NULLが戻る。
　この時、最初に確保されていた領域には何も変化がない。
注意点２：
　確保されて戻ってくる領域は、最初の領域とは移動することがある。
　（主に拡大の場合だが、仕様上は縮小の場合も無いとは保証されない）
　移動の場合、min(最初のサイズ, 新しいサイズ) までの個数のメモリ領域についてはデータは前の領域からコピーされ、残りは値が不定となる。
　移動された場合、昔の領域を指していたポインタには意味がなくなる。
注意点３：
　第一引数にNULLを与えると、malloc( ) と同じ動作をする。
注意点４：
　小さな単位でrealloc( )を繰り返すと「フラグメンテーション」の危険有り。

# 配列のサイズの再割当て

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE1   1000000
#define SIZE2   5000000

int main(void){
        int i, *ptr;

        if((ptr = calloc(SIZE1, sizeof(int))) == NULL){
                fprintf(stderr, "calloc error.¥n");
                exit(1);
        };
        for(i=0; i<SIZE1; i++){
                ptr[i] = i;
        }

        if((ptr = realloc(ptr, sizeof(int)*SIZE2)) == NULL){
                fprintf(stderr, "realloc error.¥n");
                exit(1);
        };
        for(i=SIZE1; i<SIZE2; i++){
                ptr[i] = i;
        }
        for(i=SIZE2-10; i<SIZE2; i++){
                printf("i=%d, *ptr=%d¥n", i, ptr[i]);
        }
}
```

**静的に割当てた配列ではなく、**
**calloc( ) 等で取った領域からでないと**
**realloc( )は許されない点に注意。**

**最初の１００万要素を使い尽くす**

**５００万要素に拡張してから使う**

**要素の末尾周辺を確認のため印刷**

# free( ) 関数　　メモリの解放

```
void  free(void *ptr);
```

引数には、malloc( ) 等の関数で確保した領域の先頭のポインタを与える。
free( ) 関数により、**確保されていた領域が解放**され再利用が可能となる。

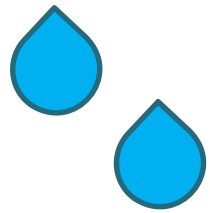
**注意点１：**
malloc( ) などで動的に確保された領域は、基本的には、必ず free ( ) をする必要があると理解しておくべきである。
（これを怠ると、後述のメモリリークの原因となる）
ただし、プログラムの終了時には自動的に解放される。

**注意点２：**
free( ) してしまった領域は、プログラムからアクセスしてはいけない。
アクセスした場合の動作は不定。

**注意点３：**
free( ) の引数に間違ったポインタを与えた場合の動作は不定。

**注意点４：**
free (NULL) はエラーではなく、何もしないで返る。

# メモリリーク (memory leak)



発生しがちなバグの一種。
メモリの動的割当てを頻繁に用いるプログラムにおいて、
確保した領域が不要になっても、それを正しく解放せず、
確保ばかり続けていくことによりメモリ不足となる状態。

プログラムを長く動作させていると利用領域が増大し、
やがてメモリ不足でプログラムが異常終了して判明する。

（発生する状況）
- 不要になった領域をfree( ) する配慮が全くない。
- 条件分岐等によりfree( ) されないケースを見逃し。
- free()に渡すべきポインタ情報が正しくない。

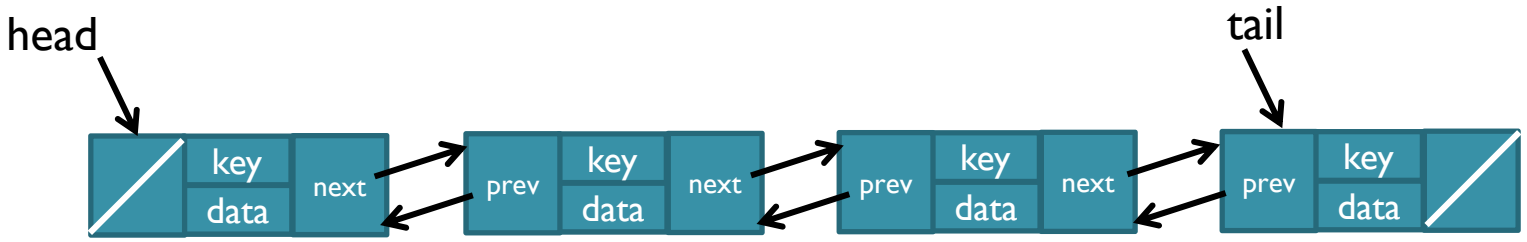
（対策）
- メモリの動的割当てを関数化するなど流れを整理し
  不要な領域については、確実に free ( ) 関数を呼ぶ。

# データ構造と動的メモリ割当て

- **配列** (array)
- **スタック** (stack)　- LIFO
- **キュー** (queue)　- FIFO
- **リスト構造** (linked list)
- **木構造** (tree)

**あらかじめ必要な要素数が判らない場合に、**
**今回の「動的メモリ割当て」の技法を用いることで、**
**データ構造の容量を増減することが可能となる。**

# リスト (linked list)の例




リスト構造は、（このプログラムの場合）entry型と呼ぶ要素を多数準備し、その間をポインタで連結したもの。メモリ上の領域を確保が必要となる。

方法１： 要素を一定個数分だけ格納できる配列を取り、自ら管理する。（先週）
その一定個数を超えたときは、要素が追加できない。

方法２：メモリの動的割り当てを用いて、必要なたびにOSに要求する。（今週）
OS側が提供できる限界を超えたときには、要素が追加できない。

２－１： メモリ動的割当ては、必要なたびに、要素１つごとに行う。
（ソースコードはわかりやすい。実行効率が悪い。）

２－２： メモリ動的割当ては、不足したときに、まとめた単位で行う。
（ソースコードがやや複雑になる。実行効率が良い。）

## 方法２－１：　要素１つごとの割当て

```
/* linked list (memory allocation one by one) */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define STRLEN 20
typedef struct entry{
    int key;
    char data[STRLEN];
    struct entry *prev;
    struct entry *next;
} entry;

entry* insert_list(entry *head, int key, char string[]);
entry* delete_list(entry *head, entry *ptr);
entry* search_list(entry *ptr, int key);
void print_list(entry *p);
void print_entry(entry *p);

int main(void){
    int i;
    entry *head;

    head = NULL;
    head = insert_list(head, 21, "KAWASHIMA");
    head = insert_list(head,  3, "KOMANO");
    head = insert_list(head, 22, "NAKAZAWA");
    head = insert_list(head,  4, "TULIO");
    head = insert_list(head,  5, "NAGATOMO");
```

**前回の例とは異なり**
**entry 型の pool をユーザは宣言しない。**
**フリーリストもユーザは宣言しない。**
**それらの初期化も当然ながら必要ない。**

```
    head = insert_list(head,  2, "ABE");
    head = insert_list(head,  8, "MATSUI");
    head = insert_list(head, 17, "HASEBE");
    head = insert_list(head,  7, "ENDO");
    head = insert_list(head, 16, "OKUBO");
    head = insert_list(head, 18, "HONDA");
    print_list(head);

    head = delete_list(head, search_list(head,  8));
    head = insert_list(head,  9, "OKAZAKI");
    print_list(head);

    head = delete_list(head, search_list(head, 16));
    head = insert_list(head, 15, "KONNO");
    print_list(head);

    head = delete_list(head, search_list(head, 7));
    head = insert_list(head, 20, "INAMOTO");

    for(i=0; i<10000000; i++){
        head = insert_list(head, 99, "NONAME");
    }
    for(i=0; i<10000000; i++){
        head = delete_list(head, search_list(head, 99));
    }
    print_list(head);
    return 0;
}
```

**試しに・・**
**1000万件連続挿入**

**1000万件連続削除**

```
entry* insert_list(entry *ptr, int key, char string[]){
    entry *new;

    /* allocation */
    if((new = calloc(1, sizeof(entry))) == NULL){
        fprintf(stderr, "calloc error.¥n");
        exit(1);
    }

    new->key = key;
    if(strlen(string) < STRLEN){
        strcpy(new->data, string);
    }else{
        fprintf(stderr, "string too long: %s¥n", string); exit(1);
    }

    /* insertion */
    new->prev = NULL;
    new->next = ptr;
    if(ptr != NULL){
        ptr->prev = new;
    }
    return(new);
}
```

entry型の１要素分を
calloc( ) 関数で確保。

```
entry* delete_list(entry *head, entry *ptr){
        if(ptr->next != NULL){
                (ptr->next)->prev = ptr->prev;
        }
        if(ptr->prev != NULL){
                (ptr->prev)->next = ptr->next;
        }else{
                head = ptr->next; /* new head */
        }

        /* garbage  collection */
        free(ptr);
        return(head);
}
```

entry型の１要素分を
free( ) 関数で解放。

```
[18,HONDA] --> [16,OKUBO] --> [7,ENDO] --> [17,HASEBE] --> [8,MATSUI] --> [2,ABE
] --> [5,NAGATOMO] --> [4,TULIO] --> [22,NAKAZAWA] --> [3,KOMANO] --> [21,KAWASH
IMA] --> NULL

[9,OKAZAKI] --> [18,HONDA] --> [16,OKUBO] --> [7,ENDO] --> [17,HASEBE] --> [2,AB
E] --> [5,NAGATOMO] --> [4,TULIO] --> [22,NAKAZAWA] --> [3,KOMANO] --> [21,KAWAS
HIMA] --> NULL

[15,KONNO] --> [9,OKAZAKI] --> [18,HONDA] --> [7,ENDO] --> [17,HASEBE] --> [2,AB
E] --> [5,NAGATOMO] --> [4,TULIO] --> [22,NAKAZAWA] --> [3,KOMANO] --> [21,KAWAS
HIMA] --> NULL

[20,INAMOTO] --> [15,KONNO] --> [9,OKAZAKI] --> [18,HONDA] --> [17,HASEBE] --> [
2,ABE] --> [5,NAGATOMO] --> [4,TULIO] --> [22,NAKAZAWA] --> [3,KOMANO] --> [21,K
AWASHIMA] --> NULL
```

開始時点

OUT 松井
IN　岡崎

OUT 大久保
IN　今野

（1000万件）

OUT 遠藤
IN　稲本

## 方法２－１：　要素１つごとの割当て

```
node* insert_tree(node *ptr, int key, char string[]){
    node *new, *x, *y;

    /* allocation */
    if((new = calloc(1, sizeof(node))) == NULL){
            fprintf(stderr, "calloc error.¥n"); exit(1);
    }
    new->key = key;
    if(strlen(string) < STRLEN){
            strcpy(new->data, string);
    }else{
            fprintf(stderr, "string too long: %s¥n", string); exit(1);
    }
    /* insertion */
    y = NULL;
    x = ptr;
    while(x != NULL){
            y = x;
            if(key < x->key){
                x = x->left;
            }else{
                x = x->right;
            }
    }
    new->parent = y;
    new->left = new->right = NULL;
    if(y == NULL){
            return(new);
    }else{
            if(key < y->key){
                y->left = new;
            }else{
                y->right = new;
            }
            return(ptr);
    }
}
```

# 木構造 (tree)の例

２－１：　メモリ動的割当ては、必要なたびに、１要素ずつ行う。
（ソースコードはわかりやすい。実行効率が悪い。）

たしかにソースコードは判りやすく、開発面では適することが判った。

２－２：　メモリ動的割当ては、不足したときに、まとめた単位で行う。
（ソースコードがやや複雑になる。実行効率が良い。）

次に２－２の方法でもプログラムを開発する。（数倍の高速化を期待）
先週、固定長の配列から自分で管理するバージョンを書いているため
比較的簡単に書けるが、２－１に比べればやや複雑である。

なお、ここでは以下の方針を採ることにする。

・不要となった要素はフリーリストに戻して有効に自己管理する。
しかしそれでもフリーリストが空になったときに、
新たな領域を（例えば１０万要素分）まとめて動的割当てして、
フリーリストを補充する。

・フリーリストが長くなっても、それを整理してOS側に free ( ) で
返す機能は実現しない。このため、free( )関数は一度も呼ばない。
プログラムのメモリ領域は一度大きくなると小さくは戻らない。

## 方法２－２：　まとめた単位で割当て

```
/* linked list (memory allocation as a chunk) */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 100000
#define STRLEN 20

typedef struct entry{
          int key;
          char data[STRLEN];
          struct entry *prev;
          struct entry *next;
} entry;

entry *freelist;

void allocate_pool(int size);
entry* insert_list(entry *head, int key, char string[]);
entry* delete_list(entry *head, entry *ptr);
entry* search_list(entry *ptr, int key);
void print_list(entry *p);
void print_entry(entry *p);

int main(void){
          int i;
          entry *head;
```

poolの容量は割当て処理のたびに10万件分

entry 型の pool は、動的に確保されるようにするため、明示的には宣言していない。フリーリストへのポインタは宣言する。

名前がfree だと、stdlib.h で定義されている free( ) 関数と干渉するために改名しました。

```
    allocate_pool(SIZE);
    head = NULL;
    head = insert_list(head, 21, "KAWASHIMA");
    head = insert_list(head,  3, "KOMANO");
```

← はじめの10万件分の領域を確保。

（途中省略：　スライド13～14と同様の11件の登録）

```
    head = delete_list(head, search_list(head,  8));
    head = insert_list(head,  9, "OKAZAKI");
    print_list(head);

    head = delete_list(head, search_list(head, 16));
    head = insert_list(head, 15, "KONNO");
    print_list(head);

    head = delete_list(head, search_list(head, 7));
    head = insert_list(head, 20, "INAMOTO");

    for(i=0; i<10000000; i++){
        head = insert_list(head, 99, "NONAME");
    }
    for(i=0; i<10000000; i++){
        head = delete_list(head, search_list(head, 99));
    }
    print_list(head);
    return 0;
}
```

1000万件連続挿入

1000万件連続削除

２－１方式よりも、数倍高速

```
void allocate_pool(int size){
     int i;
     entry *pool;

     if(size < 1){
          fprintf(stderr, "Illegal pool size %d.¥n", size); exit(1);
     }
     if(freelist != NULL) return;

     if((pool = calloc(size, sizeof(entry))) == NULL){
          fprintf(stderr, "calloc error.¥n");
          exit(1);
     }

     for(i=0; i < (size-1); i++){
          pool[i].next = &(pool[i+1]);
     }
     pool[size-1].next = NULL;
     freelist = pool;
}
```

フリーリストが空でない時は無効。

まとめて新たなpoolを確保する。
ここでは　size　は10万件。
（注：前のpoolと番地の連続性は不要）

先週と同様の初期化処理。
pool内のentryを数珠つなぎに
して、新たなフリーリストを作成。
（以前のフリーリストは既にNULL
なので、特段の処理は必要ない）

```
entry* insert_list(entry *ptr, int key, char string[]){
    entry *new;

    /* allocation */
    if(freelist == NULL){
        allocate_pool(SIZE);
    }
    new = freelist;
    freelist = new->next;

    new->key = key;
    if(strlen(string) < STRLEN){
        strcpy(new->data, string);
    }else{
        fprintf(stderr, "string too long: %s¥n", string); exit(1);
    }

    /* allocation */
    new->prev = NULL;
    new->next = ptr;
    if(ptr != NULL){
        ptr->prev = new;
    }
    return(new);
}
```

フリーリストが空のときは、
１個分ではなく、まとめて
指定個数分だけ確保する関数を呼ぶ。

フリーリストから１つの要素を
取り出してつなぎ替えるなどの
処理はすべて手動で行う。

```
entry* delete_list(entry *head, entry *ptr){
    if(ptr->next != NULL){
        (ptr->next)->prev = ptr->prev;
    }
    if(ptr->prev != NULL){
        (ptr->prev)->next = ptr->next;
    }else{
        head = ptr->next; /* new head */
    }
    /* garbage  collection */
    ptr->next = freelist;
    freelist = ptr;
    return(head);
}
```

entry型の１要素分を
手動でフリーリストに戻す。
次に再利用される。
ただしOSには返していない。
(OSに返すには、割当てた際の
単位で解放する必要があるから)

```
[18,HONDA] --> [16,OKUBO] --> [7,ENDO] --> [17,HASEBE] --> [8,MATSUI] --> [2,ABE
] --> [5,NAGATOMO] --> [4,TULIO] --> [22,NAKAZAWA] --> [3,KOMANO] --> [21,KAWASH
IMA] --> NULL

[9,OKAZAKI] --> [18,HONDA] --> [16,OKUBO] --> [7,ENDO] --> [17,HASEBE] --> [2,AB
E] --> [5,NAGATOMO] --> [4,TULIO] --> [22,NAKAZAWA] --> [3,KOMANO] --> [21,KAWAS
HIMA] --> NULL

[15,KONNO] --> [9,OKAZAKI] --> [18,HONDA] --> [7,ENDO] --> [17,HASEBE] --> [2,AB
E] --> [5,NAGATOMO] --> [4,TULIO] --> [22,NAKAZAWA] --> [3,KOMANO] --> [21,KAWAS
HIMA] --> NULL

[20,INAMOTO] --> [15,KONNO] --> [9,OKAZAKI] --> [18,HONDA] --> [17,HASEBE] --> [
2,ABE] --> [5,NAGATOMO] --> [4,TULIO] --> [22,NAKAZAWA] --> [3,KOMANO] --> [21,K
AWASHIMA] --> NULL
```

開始時点

OUT 松井
IN　岡崎

OUT 大久保
IN　今野

（1000万件）

OUT 遠藤
IN　稲本

# 【補足】main( )の引数： argc, argv

```
#include <stdio.h>
int main(int argc, char **argv){
    int x = 100; // default
    int y = 200; // default

    if(argc > 1){
        sscanf(argv[1], "%d", &x);
    }
    if(argc > 2){
        sscanf(argv[2], "%d", &y);
    }
    printf("%s %d %d¥n", argv[0], x, y);
}
```

```
c:¥temp>argtest
argtest 100 200

c:¥temp>argtest 111
argtest 111 200

c:¥temp>argtest 111 222
argtest 111 222

c:¥temp>argtest 111 222 333
argtest 111 222
```

## int argc

コマンド起動時の引数の個数が渡される。コマンド名そのものも引数の個数に含みます。

左下の実行例でいえば、引数の数はそれぞれ1個,2個, 3個,4個。

## char **argv

引数等の文字列が渡される。

char* argv[0]　コマンド名
char* argv[1]　第１引数
char* argv[2]　第２引数
char* argv[3]　第３引数

argc の値を確認してから、
argv [i]の文字列を解析すること。

（参考）getopt( ) 関数
　　オプション解析用